Arrays

- An *array* is a group of like-typed variables that are referred to by a common name
- A specific element in an array is accessed by its index

One-Dimensional Arrays:

type var-name[ ]; - No array exists

array-var = new type[size]; - allocating memory

Example: int month_days[] = new int[12]

- Arrays can be initialized when they are declared
- An *array initializer* is a list of comma-separated expressions surrounded by curly braces
- There is no need to use **new**

```
class Average {
public static void main(String args[]) {
double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
double result = 0;
int i;
for(i=0; i<5; i++)
result = result + nums[i];
System.out.println("Average is " + result / 5);
} }
```

Multidimensional Arrays

- To declare a multidimensional array variable, specify each additional index using another set of square brackets

Example:

int twoD[][] = new int[4][5];

```
class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
} } }
```

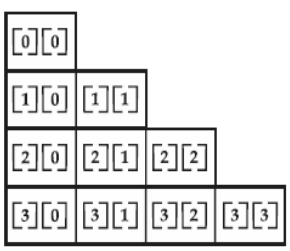
- When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension
- You can allocate the remaining dimensions separately

Example:

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

- When you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension
- Since multidimensional arrays are actually arrays of arrays, the length of each array is under your control

```
class TwoDAgain {
public static void main(String args[]) {
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<i+1; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
} } }
```



- It is possible to initialize multidimensional arrays
- You can use expressions as well as literal values inside of array initializers

```
class Matrix {
public static void main(String args[]) {
double m[ ][ ] = {
{ 0*0, 1*0, 2*0, 3*0 },
```

```
{ 0*1, 1*1, 2*1, 3*1 },
{ 0*2, 1*2, 2*2, 3*2 },
{ 0*3, 1*3, 2*3, 3*3 }
};
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<4; j++)
System.out.print(m[i][j] + " ");
System.out.println();
}}}
```

Output:

```
0.0 0.0 0.0 0.0
0.0 1.0 2.0 3.0
0.0 2.0 4.0 6.0
0.0 3.0 6.0 9.0
```

Alternative Array Declaration Syntax

- There is a second form that may be used to declare an array:

  type*[ ] *var-name;

Example: These two are equivalent

```
int al[ ] = new int[3];
int[ ] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[ ][ ] = new char[3][4];
char[ ][ ] twod2 = new char[3][4];
```

Note:

- Java does not support or allow pointers
- Java cannot allow pointers, because doing so would allow Java applets to breach the firewall between the Java execution environment and the host computer
- Java is designed in such a way that as long as you stay within the confines of the execution environment, you will never need to use a pointer, nor would there be any benefit in using one

Operators

Arithmetic Operators

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

- The modulus operator, **%**, returns the remainder of a division operation
- It can be applied to floating-point types as well as integer types
- This differs from C/C++, in which the **%** can only be applied to integer types

```
class Modulus {
public static void main(String args[]) {
int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
} }
```

When you run this program you will get the following output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Assignment Operators

- a = a + 4;
- a += 4;

Any statement of the form ***var = var op expression*;** can be rewritten as

***var op= expression*;**

The Bitwise Operators

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment

Operator	Result
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

```
class BitLogic {
public static void main(String args[]) {
String binary[ ] = {
"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
"1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
};
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
```

```
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b) | (a & ~b);
int g = ~a & 0x0f;
System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);
System.out.println(" a|b = " + binary[c]);
System.out.println(" a&b = " + binary[d]);
System.out.println(" a^b = " + binary[e]);
System.out.println("~a&b|a&~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);
}
}
```

The Left Shift

It has this general form: ***value << num***

- If you left-shift a **byte** value, that value will first be promoted to **int** and then shifted
- This means that you must discard the top three bytes of the result if what you want is the result of a shifted **byte** value
- The easiest way to do this is to simply cast the result back into a **byte**

```
class ByteShift {
public static void main(String args[]) {
byte a = 64, b;
int i;
i = a << 2;
b = (byte) (a << 2);
System.out.println("Original value of a: " + a);
System.out.println("i and b: " + i + " " + b);
} }
```

Output: ***Original value of a*: *64***
i and b: 256 0

The Right Shift

- The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times

value >> num

```
int a = 32;
a = a >> 2; // a now contains 8
```

```
11111000 –8
      >>1
11111100 –4        - Sign extension
```

```
// Masking sign extension.
class HexByte {
static public void main(String args[]) {
char hex[] = {
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
};
byte b = (byte) 0xf1;
System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
} }
```

The Unsigned Right Shift

- Unsigned, shift-right operator, **>>>**, which always shifts zeros into the high-order bit

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:

11111111 11111111 11111111 11111111 –1 in binary as an int

>>>24

00000000 00000000 00000000 11111111 255 in binary as an int

// Unsigned shifting a byte value.

```
class ByteUShift {
static public void main(String args[]) {
char hex[] = {
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
};
byte b = (byte) 0xf1;
byte c = (byte) (b >> 4);
byte d = (byte) (b >>> 4);
byte e = (byte) ((b & 0xff) >> 4);
System.out.println(" b = 0x“ + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
System.out.println(" b >> 4 = 0x“ + hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);
System.out.println(" b >>> 4 = 0x“ + hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);
System.out.println("(b & 0xff) >> 4 = 0x“ + hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);
} }
```

Output

```
b = 0xf1
b >> 4 = 0xff
b >>> 4 = 0xff
(b & 0xff) >> 4 = 0x0f
```

Relational Operators

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

```
int a = 4;        int b = 1;        boolean c = a < b;

int done;
// ...
                  if(!done) ... // Valid in C/C++
                  if(done) ... // but not in Java.

        In Java, these statements must be written like this:
                  if(done == 0)) ... // This is Java-style.
                  if(done != 0) ...
```

*In Java, **true** and **false** are nonnumeric values which do not relate to zero or nonzero*

Boolean Logical Operators

<table>
<tr><th>Operator</th><th>Result</th></tr>
<tr><td>&</td><td>Logical AND</td></tr>
<tr><td>|</td><td>Logical OR</td></tr>
<tr><td>^</td><td>Logical XOR (exclusive OR)</td></tr>
<tr><td>||</td><td>Short-circuit OR</td></tr>
<tr><td>&&</td><td>Short-circuit AND</td></tr>
<tr><td>!</td><td>Logical unary NOT</td></tr>
<tr><td>&=</td><td>AND assignment</td></tr>
<tr><td>|=</td><td>OR assignment</td></tr>
<tr><td>^=</td><td>XOR assignment</td></tr>
<tr><td>==</td><td>Equal to</td></tr>
<tr><td>!=</td><td>Not equal to</td></tr>
<tr><td>?:</td><td>Ternary if-then-else</td></tr>
</table>

Short-Circuit Logical Operators

- Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone

Example 1: if (denom != 0 && num / denom > 10)
Example 2: if(c==1 & e++ < 100) d = 100;

The Assignment Operator

var = *expression*;
int x, y, z;
x = y = z = 100; // set x, y, and z to 100

The ? Operator
expression1 **?** *expression2* **:** *expression3*

Example:
ratio = denom == 0 ? 0 : num / denom;

Operator Precedence

Highest			
( )	[ ]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

Example:
a | 4 + c >> b & 7
(a | (((4 + c) >> b) & 7))

- ***Parentheses (redundant or not) do not degrade the performance of your program***
- ***Therefore, adding parentheses to reduce ambiguity does not negatively affect your program***

Control Statements
If:
if (*condition*) *statement1*;
else *statement2*;

Nested If:

```
if(i == 10) {
    if(j < 20) a = b;
    if(k > 100) c = d; // this if is
    else a = c; // associated with this else
    }
else a = d; // this else refers to if(i == 10)
```

<u>The if-else-if Ladder:</u>

if(*condition*)
statement;
else if*(condition*)
statement;
else if(*condition*)
statement;
...
else
statement;

<u>switch</u>

switch (*expression*) {
case *value1*:
// statement sequence
break;
case *value2*:
// statement sequence
break;
...
case *valueN*:
// statement sequence
break;
default:
// default statement sequence
}

- ***The expression must be of type byte, short, int, or char;***

<u>Nested switch Statements</u>

```
switch(count) {
case 1:
switch(target) { // nested switch
case 0:
System.out.println("target is zero");
break;
case 1: // no conflicts with outer switch
System.out.println("target is one");
break;
```

```
}
break;
case 2: // ...
```

<u>Iteration Statements</u>

While:
```
while(condition) {
        // body of loop
    }
```

<u>do-while</u>
```
do {
        // body of loop
} while (condition);
```

<u>for</u>
```
for(initialization; condition; iteration) {
// body
}
```

<u>Some for Loop Variations</u>

```
boolean done = false;
for(int i=1; !done; i++) {
        // ...
if(interrupted()) done = true;
}
```

<u>Jump Statements</u>

Using break to Exit a Loop:

```
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<100; i++) {
if(i == 10) break; // terminate loop if i is 10
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
} }
```

- More than one **break** statement may appear in a loop
- Too many **break** statements have the tendency to destructure your code
- The **break** that terminates a **switch** statement affects only that **switch** statement and not any enclosing loops

<u>Using break as a Form of Goto</u>

- Java defines an expanded form of the **break** statement
- By using this form of **break**, you can break out of one or more blocks of code

- The general form of the labeled **break** statement is : **break *label*;**
- A *label* is any valid Java identifier followed by a colon
- You can use a labeled **break** statement to exit from a set of nested blocks
- You cannot use **break** to transfer control to a block of code that does not enclose the **break** statement

```
class Break {
public static void main(String args[]) {
boolean t = true;
first: {
second: {
third: {
System.out.println("Before the break.");
if(t) break second; // break out of second block
System.out.println("This won't execute");
}
System.out.println("This won't execute");
}
System.out.println("This is after second block.");
} } }
```

```
class BreakLoop4 {
public static void main(String args[]) {
outer: for(int i=0; i<3; i++) {
System.out.print("Pass " + i + ": ");
for(int j=0; j<100; j++) {
if(j == 10) break outer; // exit both loops
System.out.print(j + " ");
}
System.out.println("This will not print");
}
System.out.println("Loops complete.");
} }
```

```
// This program contains an error.
class BreakErr {
public static void main(String args[]) {
one: for(int i=0; i<3; i++) {
System.out.print("Pass " + i + ": ");
}
for(int j=0; j<100; j++) {
if(j == 10) break one; // WRONG
System.out.print(j + " ");
} } }
```

Using continue

```
class Continue {
public static void main(String args[]) {
for(int i=0; i<10; i++) {
System.out.print(i + " ");
if (i%2 == 0) continue;
System.out.println("");
} } }
```

- As with the **break** statement, **continue** may specify a label to describe which enclosing loop to continue

```
class ContinueLabel {
public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
for(int j=0; j<10; j++) {
if(j > i) {
System.out.println();
continue outer;
}
System.out.print(" " + (i * j));
}
}
System.out.println();
} }
```

Output:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

return

- Used to explicitly return from a method
- The **return** statement immediately terminates the method in which it is executed

return *causes execution to return to the Java run-time system, since it is the run-time system that calls* ***main( )***